Real-Time Linux*

Behdad Esfahbod
behdad@behdad.org

Computer Engineering Department
Sharif University of Technology
Tehran, Iran

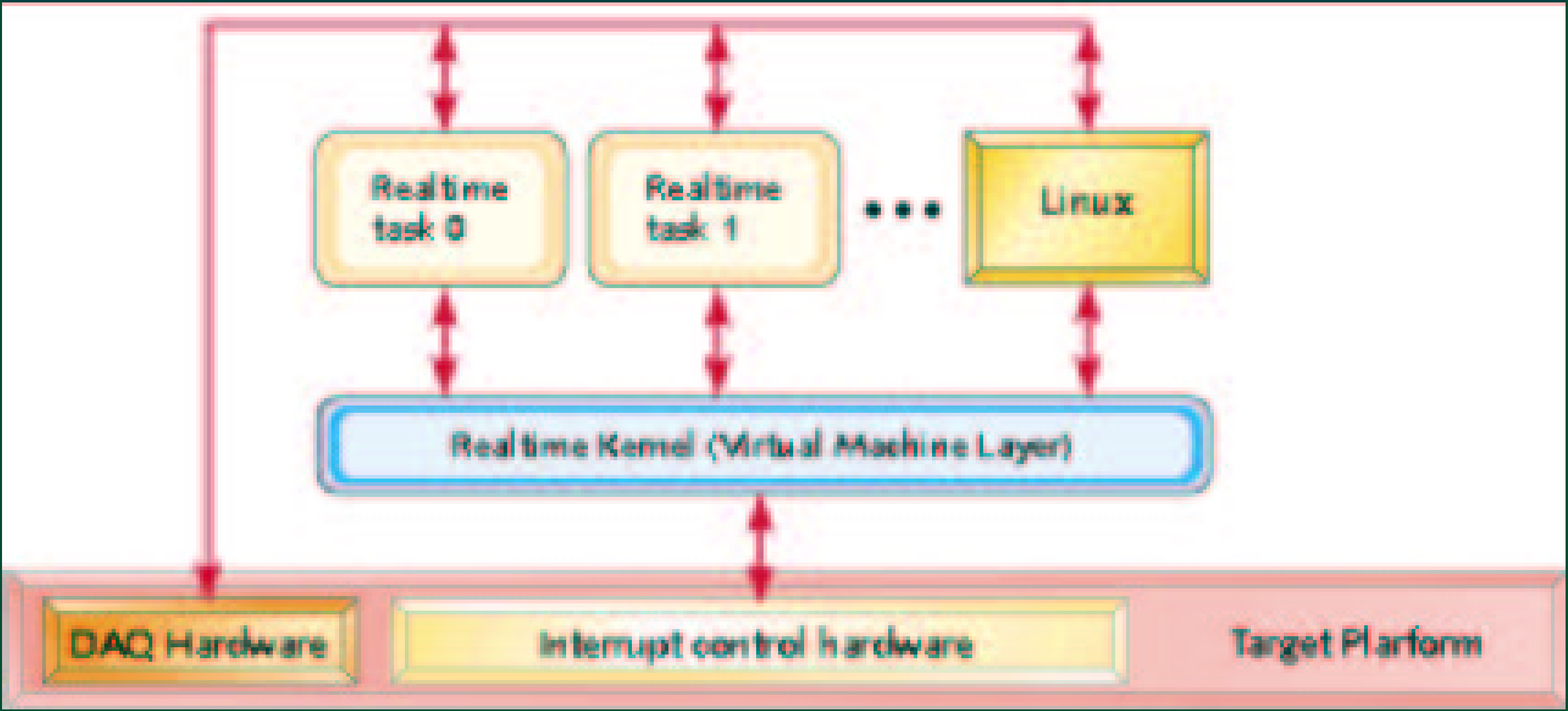
December 28, 2002

*by Alex Ivchenko, Embedded Systems Programming (May 1, 2001)
available from http://www.embedded.com/story/OEG20010418S0044
slides on http://behdad.org/presentation/rtlinux/

Main Loop

```
while (alive)
{
  get_sensor_data();  // read one or more sensors
  compute_control();  // calculate control parameters based on
             // incoming data
  control_device();   // initiate control activity
             // read next group of
             // sensors...calculate...control
}
```

Figure 1: Real-time Linux executes real-time tasks



makefile.pp_flip

```
# Makefile.pp_flip
all: pp_flip.o

RTLINUX = /usr/src/linux   # the path to the rt-linux kernel
INCLUDE = ${RTLINUX}/include
CFLAGS = -O2 -Wall

pp_flip.o: pp_flip.c
  gcc -I${INCLUDE} -I/usr/include/rtlinux ${CFLAGS} -D__KERNEL__ \
      -D__RT__  -DMODULE -c pp_flip.c

clean:
  rm -f pp_flip.o
```

Includes and Defines for pp_flip.c

```
#include <linux/module.h>
#include <linux/kernel.h>
#include <linux/version.h>
#include <linux/errno.h>
#include <rtlinux/rtl_sched.h>
#include <asm/io.h>

#define LPT1_DATA 0x378     // parallel port 1 data
#define FRQ 1000            // thread period in Hz...
#define FRAME_PERIOD_NS ((hrtime_t)((1.0/FRQ) * 1000000000.0))
                            // ...and in ns
```

pp_flip.c

```
pthread_t pp_thread;
// our periodic thread
void *pp_thread_ep(void *rate) {
    static int nState = 0;

    // make this realtime thread periodic
    pthread_make_periodic_np(pthread_self(),gethrtime(),FRAME_PERIOD_NS);
    // this loop wakes up once per period
    while (1)
    {
        if (nState)
          outb(nState, LPT1_DATA);
        else
          outb(nState++,LPT1_DATA);
        // wait until next period of time
        pthread_wait_np();
    }
}

int init_module(void) {
    pthread_attr_t attrib;
    struct sched_param sched_param;
    // output string to /var/log/kern.log
    printk(init_module pp_flip\n);

    // prepare periodic thread for creation
    // initialize thread attributes
    sched_param.sched_priority = sched_get_priority_max(SCHED_FIFO);
    // obtain highest priority
    pthread_attr_init(&attrib);
    // set our priority
    pthread_attr_setschedparam(&attrib, &sched_param);

    // and finally create the thread
    pthread_create(&pp_thread, &attrib, pp_thread_ep, (void *)0);
    return 0;
}

int cleanup_module(void) {

    printk(cleanup_module pp_flip\n);
    pthread_delete_np(pp_thread);      // kill the thread
    return 0;
}
```

Registering a Driver for Use by Real-Time and Non-Real-Time Tasks

```
// Linux driver operations
static struct file_operations ln_pd_fops =
{
read: ln_pd_read,
write: ln_pd_write,
... other magic functions
};

// RTLinux driver operations
static struct rtl_file_operations rtl_pd_fops = {
      NULL,
rtl_pd_read,
rtl_pd_write,
... other magic functions
};
```

```
// register Linux driver...
if (register_chrdev(PD_LN_MAJOR, pdaq, &ln_pd_fops))
{ handle errors... }
// ...and RTLinux driver
if (rtl_register_chrdev(PD_RT_MAJOR, pdaq, &rtl_pd_fops))
{ handle errors... }
```

Figure 2: Dual data-acquisition driver interfaces

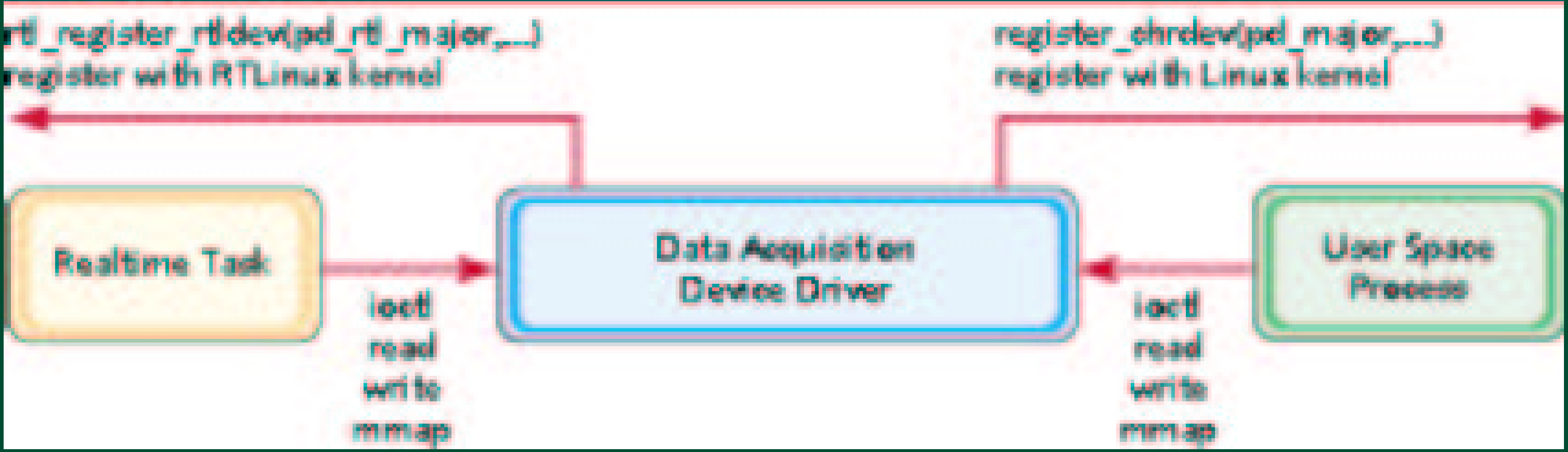
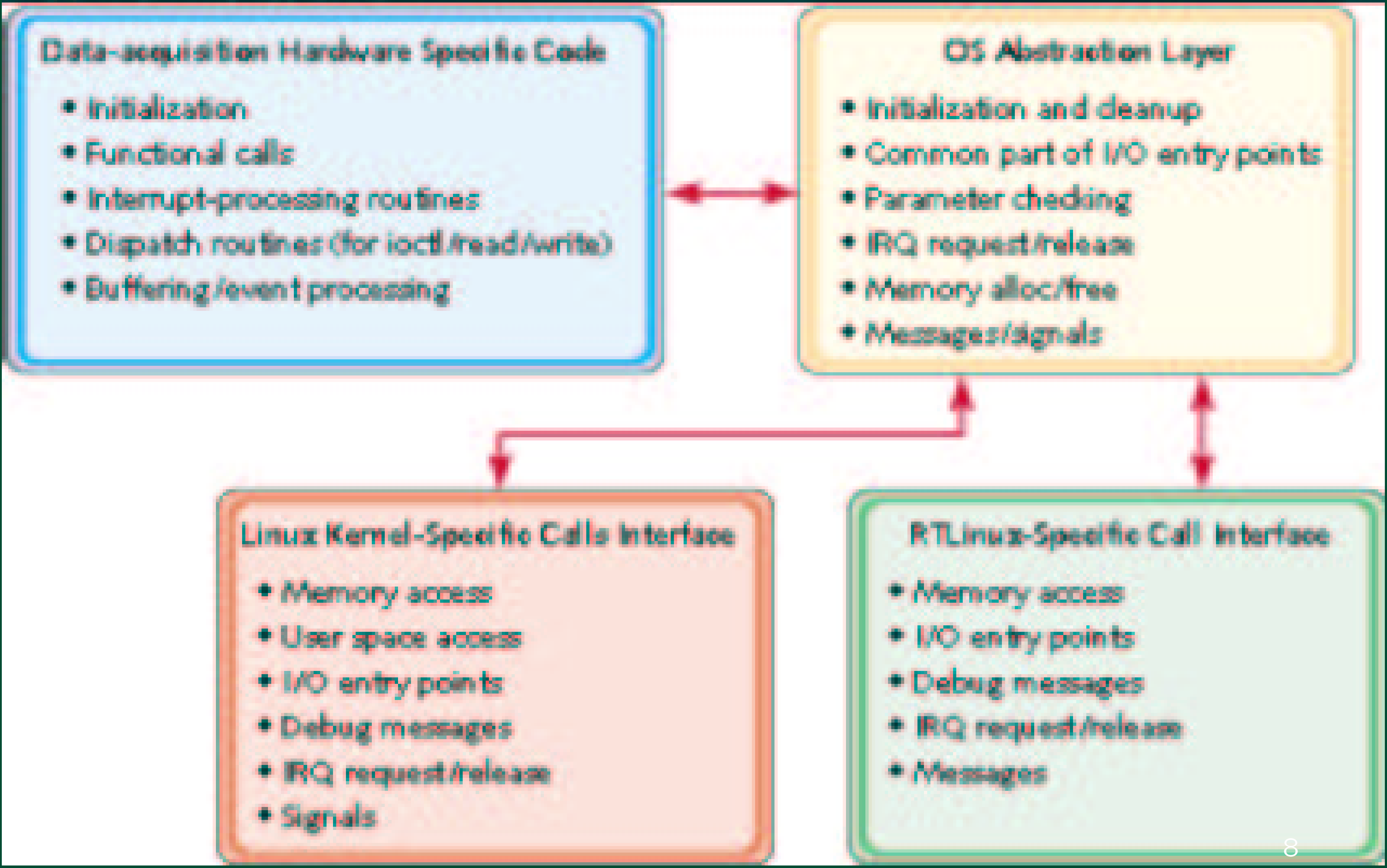

rtl_register_rtldev(pd_rtl_major,...)
register with RTLinux kernel
register_chrdev(pd_major,...)
register with Linux kernel
Realtime Task
Data Acquisition Device Driver
User Space Process
ioctl
read
write
mmap
ioctl
read
write
mmap

Figure 3: OSAL driver design allows you to run the same driver for user-space and real-time tasks



Driver's RTLinux and Linux Entry Points for read()

```
// read entry point registered by rtl_register_rtldev()
static ssize_t rtl_pd_read(struct rtl_file *filp, char *buf,
   size_t count, loff_t* ppos)
{
   u32 minor = RTL_MINOR_FROM_FILEPTR(filp);
   board = minor / PD_MINOR_RANGE;
   subsystem = minor % PD_MINOR_RANGE;
return pd_read(minor, buf, count);
}

// read entry point registered by register_chrdev()
static ssize_t ln_pd_read(struct file *filp, char *buf,
   size_t len, loff_t* ppos)
{
u32 minor = MINOR(inode->i_rdev);
return pd_read(minor, buf, count);
}

// main read() function
int pd_read(u32 minor, char *inpbuf, size_t count)
{
   u32 board = minor / PD_MINOR_RANGE;
   u32 subsystem = minor % PD_MINOR_RANGE;
      // process read request to particular board and subsystem
      ...
}
```

Different Calls for User and Kernel Spaces

```
unsigned long osal_memcpy32(u32* to, u32* from, u32 len)
{
#ifdef _NO_USERSPACE
     return (u32)memcpy(to, from, len);
#else
     return copy_from_user(to, from, len);
#endif
}
```